关于 NodeJS 中的 Process 你了解多少？

关于 Process，我们需要讨论的是两个概念，①操作系统的进程，② Node.js 中的 Process 对象. 操作进程对于服务端而言，好比 html 之于前端一样基础. 想做服务端编程是不可能绕过 Unix/Linux 的. 在 Linux/Unix/Mac 系统中运行 `ps -ef` 命令可以看到当前系统中运行的进程. 各个参数如下:

列名称	意义
UID	执行该进程的用户 ID
PID	进程编号
PPID	该进程的父进程编号
C	该进程所在的 CPU 利用率
STIME	进程执行时间
TTY	进程相关的终端类型
TIME	进程所占用的 CPU 时间
CMD	创建该进程的指令

关于进程以及操作系统一些更深入的细节推荐阅读 APUE，即《Unix 高级编程》等书籍来了解.

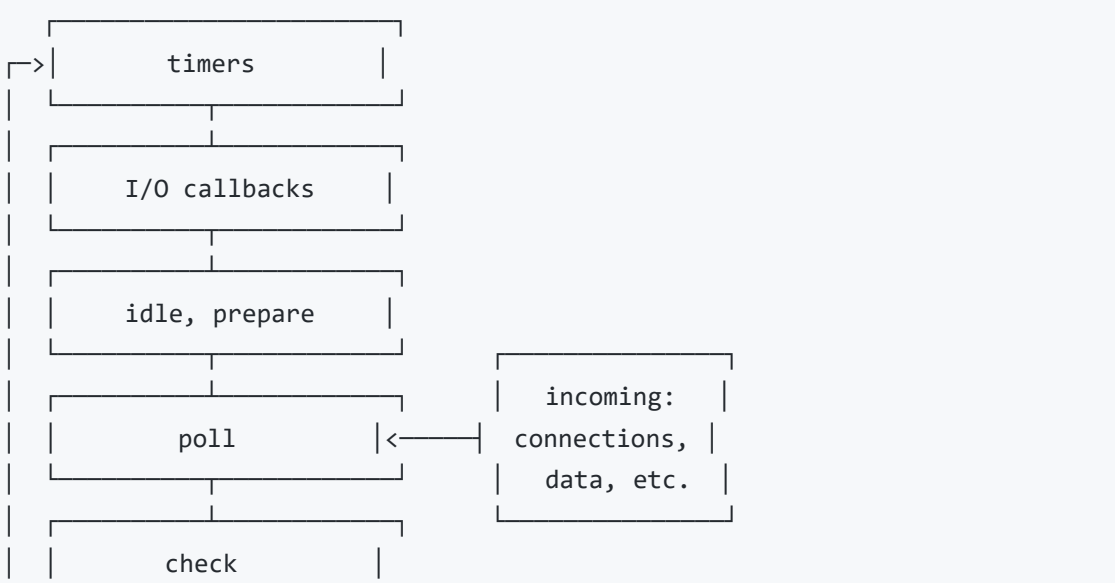
Process

这里来讨论 Node.js 中的 process 对象. 直接在代码中通过 console.log(process) 即可打印出来. 可以看到 process 对象暴露了非常多有用的属性以及方法, 具体的细节见官方文档, 已经说的挺详细了. 其中包括但不限于:

- 进程基础信息
- 进程 Usage
- 进程级事件
- 依赖模块/版本信息
- OS 基础信息
- 账户信息
- 信号收发
- 三个标准流

process.nextTick

这是一个你需要了解的, 重要的, 基础方法.



```
┌   └─────────────┬──────────────┘
│   ┌─────────────┴──────────────┐
└──┤    close callbacks         │
   └────────────────────────────┘
```

process.nextTick 并不属于 Event loop 中的某一个阶段，而是在 Event loop 的每一个阶段结束后，直接执行 nextTickQueue 中插入的 "Tick"，并且直到整个 Queue 处理完. 所以面试时又有可以问的问题了，递归调用 process.nextTick 会怎么样? (doge

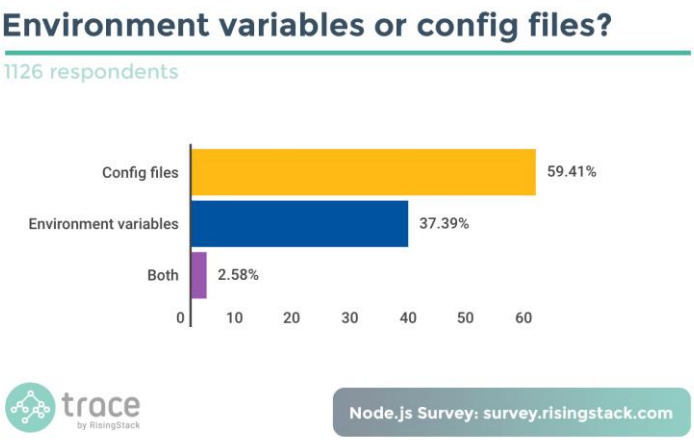
```
function test() {
  process.nextTick(() => test());
}
```

这种情况与以下情况，有什么区别？为什么？

```
function test() {
  setTimeout(() => test(), 0);
}
```

配置

配置是开发部署中一个很常见的问题. 普通的配置有两种方式，一是定义配置文件，二是使用环境变量.



你可以通过设置环境变量来指定配置，然后通过 `process.env` 来获取配置项. 另外也可以通过读取定义好的配置文件来获取，在这方面有很多不错的库例如 `dotenv`, `node-config` 等，而在使用这些库来加载配置文件的时候，通常都会碰到一个当前工作目录的问题.

进程的当前工作目录是什么？有什么作用?

当前进程启动的目录，通过 process.cwd() 获取当前工作目录 (current working directory)，通常是命令行启动的时候所在的目录 (也可以在启动时指定)，文件操作等使用相对路径的时候会相对当前工作目录来获取文件.

一些获取配置的第三方模块就是通过你的当前目录来找配置文件的. 所以如果你错误的目录启动脚本，可能没法得到正确的结果. 在程序中可以通过 `process.chdir()` 来改变当前的工作目录.

标准流

在 process 对象上还暴露了 `process.stderr`, `process.stdout` 以及 `process.stdin` 三个标准流，熟悉 C/C++/Java 的同学应该对此比较熟悉. 关于这几个流，常见的面试问题是问 **console.log 是同步还是异步？如何实现一个 console.log?**

如果简历中有出现 C/C++ 关键字，一般都会问到如何实现一个同步的输入 (类似实现 C 语言的 `scanf`, C++ 的 `cin`, Python 的 `raw_input` 等).

维护方面

熟悉与进程有关的基础命令，如 top, ps, pstree 等命令.

Child Process

子进程 (Child Process) 是进程中一个重要的概念. 你可以通过 Node.js 的 `child_process` 模块来执行可执行文件, 调用命令行命令, 比如其他语言的程序等. 也可以通过该模块来将 .js 代码以子进程的方式启动. 比较有名的网易的分布式架构 pomelo 就是基于该模块 (而不是 `cluster`) 来实现多进程分布式架构的.

child_process.fork 与 POSIX 的 fork 有什么区别?

Node.js 的 `child_process.fork()` 在 Unix 上的实现最终调用了 POSIX fork(2), 而 POSIX 的 fork 需要手动管理子进程的资源释放 (waitpid), child_process.fork 则不用关心这个问题, Node.js 会自动释放, 并且可以在 option 中选择父进程死后是否允许子进程存活.

- spawn() 启动一个子进程来执行命令
    - options.detached 父进程死后是否允许子进程存活
    - options.stdio 指定子进程的三个标准流
- spawnSync() 同步版的 spawn, 可指定超时, 返回的对象可获得子进程的情况
- exec() 启动一个子进程来执行命令, 带回调参数获知子进程的情况, 可指定进程运行的超时时间
- execSync() 同步版的 exec(), 可指定超时, 返回子进程的输出 (stdout)
- execFile() 启动一个子进程来执行一个可执行文件, 可指定进程运行的超时时间

- execFileSync() 同步版的 execFile()，返回子进程的输出，如何超时或者 exit code 不为 0，会直接 throw Error
- fork() 加强版的 spawn()，返回值是 ChildProcess 对象可以与子进程交互

其中 exec/execSync 方法会直接调用 bash 来解释命令，所以如果有命令有外部参数，则需要注意被注入的情况.

child.kill 与 child.send

常见会问的面试题，如 `child.kill` 与 `child.send` 的区别. 二者一个是基于信号系统，一个是基于 IPC.

父进程或子进程的死亡是否会影响对方？什么是孤儿进程?

子进程死亡不会影响父进程，不过子进程死亡时（线程组的最后一个线程，通常是"领头"线程死亡时），会向它的父进程发送死亡信号. 反之父进程死亡，一般情况下子进程也会随之死亡，但如果此时子进程处于可运行态、僵死状态等等的话，子进程将被进程 `1`（init 进程）收养，从而成为孤儿进程. 另外，子进程死亡的时候（处于"终止状态"），父进程没有及时调用 `wait()` 或 `waitpid()` 来返回死亡进程的相关信息，此时子进程还有一个 `PCB` 残留在进程表中，被称作僵尸进程.

Cluster

Cluster 是常见的 Node.js 利用多核的办法. 它是基于 `child_process.fork()` 实现的，所以 cluster 产生的进程之间是通过 IPC 来通信的，并且它也没有拷贝

父进程的空间，而是通过加入 cluster.isMaster 这个标识，来区分父进程以及子进程，达到类似 POSIX 的 fork 的效果.

```
const cluster = require('cluster');           // | |
const http = require('http');                 // | |
const numCPUs = require('os').cpus().length;  // | |    都执行了
                                              // | |
if (cluster.isMaster) {                       // |-|-----------------
  // Fork workers.                            //   |
  for (var i = 0; i < numCPUs; i++) {         //   |
    cluster.fork();                           //   |
  }                                           //   | 仅父进程执行 (a.js)
  cluster.on('exit', (worker) => {            //   |
    console.log(`${worker.process.pid} died`); //   |
  });                                         //   |
} else {                                      // |-------------------
  // Workers can share any TCP connection     // |
  // In this case it is an HTTP server        // |
  http.createServer((req, res) => {           // |
    res.writeHead(200);                       // |   仅子进程执行 (b.js)
    res.end('hello world\n');                 // |
  }).listen(8000);                            // |
}                                             // |-------------------
                                              // | |
console.log('hello');                         // | |    都执行了
```

在上述代码中 numCPUs 虽然是全局变量但是，在父进程中修改它，子进程中并不会改变，因为父进程与子进程是完全独立的两个空间. 他们所谓的共有仅仅只是都执行了，并不是同一份.

你可以把父进程执行的部分当做 a.js，子进程执行的部分当做 b.js，你可以把他们想象成是先执行了 node a.js 然后 cluster.fork 了几次，就执行了几次 node b.js. 而 cluster 模块则是二者之间的一个桥梁，你可以通过 cluster 提供的方法，让其二者之间进行沟通交流.

How It Works

worker 进程是由 child_process.fork() 方法创建的，所以可以通过 IPC 在主进程和子进程之间相互传递服务器句柄.

cluster 模块提供了两种分发连接的方式.

第一种方式 (默认方式，不适用于 windows)，通过时间片轮转法（round-robin）分发连接. 主进程监听端口，接收到新连接之后，通过时间片轮转法来决定将接收到的客户端的 socket 句柄传递给指定的 worker 处理. 至于每个连接由哪个 worker 来处理，完全由内置的循环算法决定.

第二种方式是由主进程创建 socket 监听端口后，将 socket 句柄直接分发给相应的 worker，然后当连接进来时，就直接由相应的 worker 来接收连接并处理.

使用第二种方式时理论上性能应该较高，然后时间上存在负载不均衡的问题，比如通常 70% 的连接仅被 8 个进程中的 2 个处理，而其他进程比较清闲.

进程间通信

IPC (Inter-process communication) 进程间通信技术. 常见的进程间通信技术列表如下:

z 类型	无连接	可靠	流控制	优先级
普通 PIPE	N	Y	Y	N
命名 PIPE	N	Y	Y	N

消息队列	N	Y	Y	N
信号量	N	Y	Y	Y
共享存储	N	Y	Y	Y
UNIX 流 SOCKET	N	Y	Y	N
UNIX 数据包 SOCKET	Y	Y	N	N

Node.js 中的 IPC 通信是由 libuv 通过管道技术实现的，在 windows 下由命名管道（named pipe）实现也就是上表中的最后第二个, *nix 系统则采用 UDS (Unix Domain Socket) 实现.

普通的 socket 是为网络通讯设计的，而网络本身是不可靠的，而为 IPC 设计的 socket 则不然，因为默认本地的网络环境是可靠的，所以可以简化大量不必要的 encode/decode 以及计算校验等，得到效率更高的 UDS 通信.

如果了解 Node.js 的 IPC 的话，可以问个比较有意思的问题

在 IPC 通道建立之前，父进程与子进程是怎么通信的？如果没有通信，那 IPC 是怎么建立的?

这个问题也挺简单，只是个思路的问题. 在通过 child_process 建立子进程的时候，是可以指定子进程的 env (环境变量) 的. 所以 Node.js 在启动子进程的时候，主进程先建立 IPC 频道，然后将 IPC 频道的 fd (文件描述符) 通过环境变

量（NODE_CHANNEL_FD）的方式传递给子进程，然后子进程通过 fd 连上 IPC 与父进程建立连接.

最后于进程间通信（IPC）的问题，一般不会直接问 IPC 的实现，而是会问什么情况下需要 IPC，以及使用 IPC 处理过什么业务场景等.

守护进程

最后的守护进程，是服务端方面一个很基础的概念了. 很多人可能只知道通过 pm2 之类的工具可以将进程以守护进程的方式启动，却不了解什么是守护进程，为什么要用守护进程. 对于水平好的同学，我们是希望能了解守护进程的实现的.

普通的进程，在用户退出终端之后就会直接关闭. 通过 & 启动到后台的进程，之后会由于会话（session 组）被回收而终止进程. 守护进程是不依赖终端（tty）的进程，不会因为用户退出终端而停止运行的进程.

```
// 守护进程实现（C 语言版本）
void init_daemon()
{
    pid_t pid;
    int i = 0;

    if ((pid = fork()) == -1) {
        printf("Fork error !\n");
        exit(1);
    }

    if (pid != 0) {
        exit(0);        // 父进程退出
    }

    setsid();           // 子进程开启新会话，并成为会话首进程和组长进程
    if ((pid = fork()) == -1) {
```

```
        printf("Fork error !\n");
        exit(-1);
    }
    if (pid != 0) {
        exit(0);         // 结束第一子进程，第二子进程不再是会话首进程
                        // 避免当前会话组重新与 tty 连接
    }
    chdir("/tmp");      // 改变工作目录
    umask(0);           // 重设文件掩码
    for (; i < getdtablesize(); ++i) {
       close(i);        // 关闭打开的文件描述符
    }

    return;
}
```